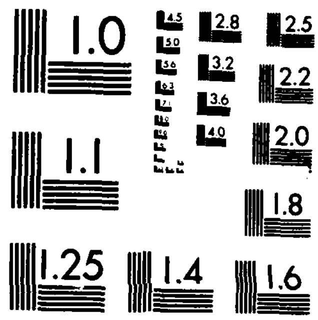
AD-A185 579 COMMUNICATIONS FOR THE DIROLL DISTRIBUTED DATABASE SYSTEM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH M F JERVIS 1986 AFIT/CI/NR-87-70T 1/1

UNCLASSIFIED F/G 12/7 NL

END
DATE
FILMED
12 87



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A185 579

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR-87-70T	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Communications For The Dtroll Distributed Database System		5. TYPE OF REPORT & PERIOD COVERED THESIS/~~DISSERTATION~~
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Michael Frank Jervis, Sr.		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: University of Illinois		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433-6583		12. REPORT DATE 1986
		13. NUMBER OF PAGES 46
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

LYNN E. WOLAVER 17 Aug 87
Dean for Research and Professional Development
AFIT/NR

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

87 10 20 173

COMMUNICATIONS FOR THE DTROLL DISTRIBUTED DATABASE SYSTEM

BY

MICHAEL FRANK JERVIS SR.

B.A., University of Texas at Austin, 1981

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science
in the Graduate College of the
University of Illinois at Urbana–Champaign, 1986

ACKNOWLEDGEMENTS

I would like to thank, my advisor, Professor Belford for her assistance in this project and for her support and advice during my time at the University. I would like to thank my colleagues on the project for the many informal papers and descriptions written, although not listed in the reference section, and sharing their ideas and problems with me. They were very instrumental in my comprehension of the system. I would also like to thank Paul Richards for the coding examples he provided. I wish to thank my wife, Arline, for her support and my children, Michael and Tony, for their cooperation and sacrifices. Finally, I greatly appreciate the United States Air Force giving me the opportunity to pursue an advanced degree and participate in this research.

Table of Contents

[BrMR,82] D.R. Brownridge, L.F. Marshall and B. Randell, The Newcastle Connection or UNIXes of the World Unite!. Software Practice and Experience (1982) vol 12, pp. 1147–1162.

[KWRi,82] M. L. Kersten, A. I. Wasserman and R. P. Riet. Troll/USE Reference Manual. Wiskundig Seminarium Vrije Universiteit de Boelelaan, Amsterdam, the Netherlands, (1982).

[KeWa,81] M. L. Kersten and A. I. Wasserman. The Architecture of the PLAIN Data Base Handler. Software — Practice and Experiences, vol 11, no 2 (February 1981), pp. 175–186.

[KeRi,78] B. W. Kernighan and D. M. Ritchie. The C Programming Language. Prentice Hall, Englewood Cliffs, NJ (1978).

INTRODUCTION

This report is intended to describe communications concepts for the **DTroll** [JKim,85] & [Mage,86] distributed database system. The report addresses the subject matter with the assumption that the reader has a basic background knowledge of UNIX (UNIX is a trademark of Bell Laboratories). For a detailed and in-depth understanding of the background and concepts surrounding the **DTroll** system, the reader is encouraged to obtain the references listed at the end of this report. In addition, informal documents are available through the project office.

This paper is presented in three parts. **Chapter 1** discusses the background and general structure of **Dtroll**, as well as establishing some of the foundation for the remainder of the paper. **Chapter 2** describes the software module for establishing a server's communications. **Chapter 3** describes the software module for establishing a client's communications. Due to the similarity of **INITSERVSOCKET** and **RQSTCONNECT**, **Chapter 2** and **Chapter 3** contain a significant amount of repetitious information. This is intentional to provide continuity and ease of understanding.

CHAPTER 1

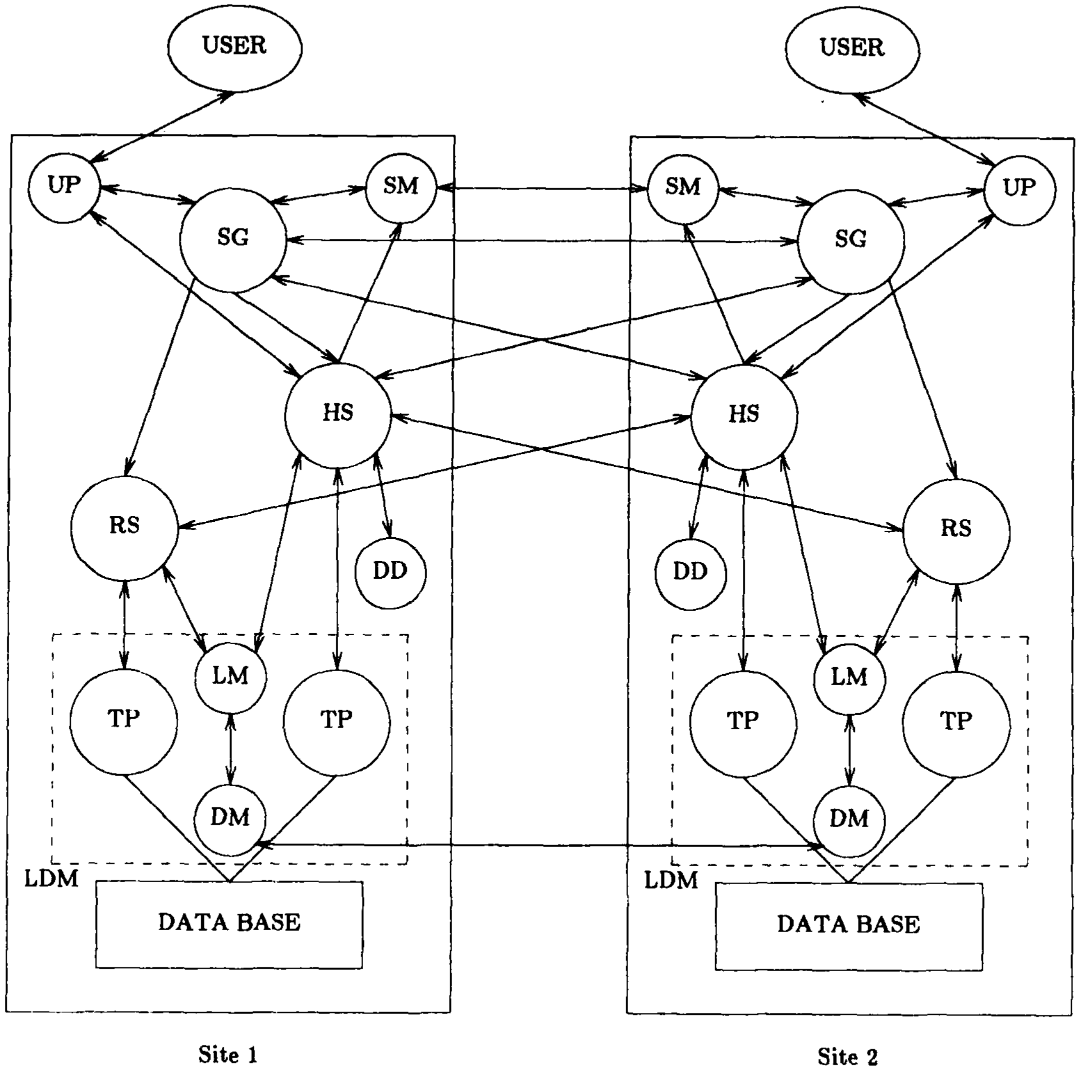
BACKGROUND

DTroll is a distributed database system which consists of a layer of functional software which, using **Troll**, [KeWa,82] & [KeWa,81], as its underlying database system, will provide a user a transparent interface to database files existing on differing hosts. The operating environment of the system is currently restricted to the VAX computers of the UIUC Computer Science Department network (VAX is a trademark of the Digital Equipment Corporation). These hosts are running **UNIX 4.2 BSD**; developed at The University of California, Berkeley; and interconnected with an Ethernet. Because the system uses features particular to the 4.2 BSD operating software [UNIX,84], the system may require extensive modification to adapt it to another **UNIX** based system.

The system will eventually be production quality. However, at its inception the stress is to get a prototype test bed system by year's end which will offer individuals the opportunity to test and experiment with differing concepts for communications, concurrency control, locking schemes, and other database concepts within a distributed arena. All software has or is being developed using the C programming language [KeRi,78].

1.1. DTroll Architecture

The overall structure of **DTroll** is depicted in figure 1.



SG : Server Generator
SM : Status Monitor
HS : Host Server
RS : Remote Server
LM : Lock Manager
TP : Troll Processor
LDM : Local Data Manager
UP : User Process
DM : Deadlock Manager
DD : Data Dictionary
↔ : Communication
— : Database Access

Figure 1: Architecture of the DTroll System

USER represents the actual user at their terminal. A user invokes the system by entering "**dtroll**" through the keyboard.

User Process (**UP**) is a process which acts as an interface from the actual user's terminal to the system. It functions as a pseudo terminal. This approach was selected to simplify the process of communicating to the user's terminal. **UP** maintains two communications channels: one to the actual terminal and another to the **Server Generator/Host Server** (see **Server Generator** for details). **UP** determines communications requests by polling the channels through the **select** system call.

Server Generator (**SG**) is responsible for spawning servers to handle user requests. Upon receipt of a request, **SG** determines if a local server (**HS**) or a remote server (**RS**) is necessary to fulfill the request. The **SG** creates a **RS** if the request is from a remote **HS** and a **HS** if the request is from a local source. Information is sent through the **UP** at the requesting site. All requests which alter data (write or modify) are handled with a two phase commit scheme to prevent the loss of data in update. This is done to guarantee all locations are consistent.

Since either a **HS** or a **RS** is created through the **fork** command, an abnormal termination of a process can be detected through the **signal** facilities. This permits the system to take any actions deemed appropriate. Once a server is spawned, the parent process is reconfigured to accept a new request.

Data Dictionary (**DD**) contains the information regarding the locations and composition (component relations) of the databases created and accessible to **DTroll**. It communicates with the **HS** to convey the existence of data or the validity of requests.

Status Monitor (SM) is used to monitor the status of the system. It has the responsibility for restarting a local **SG** in the event the process dies. It also informs the system (primarily **Host Servers**) of the status of other sites (up or down). The **Status Monitor** communicates with like processes on all the machines to determine the status of both the overall site and, with the help of Mother (an overall system monitor not shown in figure 1), the major processes of the **DTroll** system.

As stated earlier in the paper, the underlying database manipulation mechanism is **Troll**. A **Troll Process** (TP) is activated to accomplish the actual request.

Deadlock Manager (DM) is a process local to each site which maintains some sort of graph of resources requested and allocated to detect and resolve deadlock situations. It communicates with the **Lock Manager** (explained below) and remote **Deadlock Managers** to accomplish its task.

Lock Manager (LM) has the responsibility of performing all actions involved in processing locks on the relations involved in a transaction. Requests for locks come from either a **HS** or **RS**.

The **Local Data Manager** consists of those processes (**TP, LM, and DM**) which are directly involved with data management.

As one can see, with all the communications required, some provision is necessary for the setup and teardown of the various communications channels. For those familiar with **UNIX**, **pipes** may come to mind. A **pipe** is a communications circuit or mechanism for passing information between two processes. Information is written to one end of the **pipe** and read from the other end. A major drawback of a **pipe** is that the

processes must be related. That is, one of the processes must have been created by the **fork** system call issued by the other process. As the **DTroll** modules are independent, autonomous processes, a different method was required.

4.2 BSD UNIX provides **sockets** which are analogous to **pipes** with two distinct differences. First, the processes need not be related. Second, **sockets** are bidirectional. **Sockets**, like **pipes**, provide a mechanism through which information may be passed between processes. One of the drawbacks to the use of **sockets** is that one must execute a series of system calls to obtain the necessary addressing information and prepare the **socket** for use.

The routines **INITSERVSOCKET** and **RQSTCONNECT** were developed to minimize the effort necessary to utilize the **socket** facilities. Both routines offer a user wide flexibility in establishing a communications bridge between two independent processes by greatly reducing their responsibility in forming legal addresses and implementing conditioning calls. Servers accomplish the setup of a communications channel with a call to **INITSERVSOCKET** and the **accept** system call. Clients perform setup with a call to **RQSTCONNECT**. A **socket** must be established by a server process before the **RQSTCONNECT** call will be successful. Teardown of any socket is done with the **close** system call.

1.2. UNIX United as an Alternative

In considering how the communications for the **DTroll** system would be handled, close scrutiny was made of and careful consideration given to the **UNIX United** software system [Donn,85], [Goth,85], & [BrMR,82].

Basically **UNIX United** acts as an interceptor within the host on which it is activated. It intercepts commands from a user and then determines if the request should be passed on to the **Kernel** or handled as a remote operation. The software is very complicated and handles a wide variety of tasks not associated with communications. The complexity makes the system prone to failure and slower than our chosen method of communication.

One of the drawbacks of the **UNIX United** system is that all software at a supported site is not automatically supported. Systems must be linked to the **UNIX United** system in order to take advantage of its facilities. Maintenance and control of **UNIX United** and which systems are linked to it are totally outside the control of any group wishing to incorporate **UNIX United** in the design of their system. This is a serious limitation for developing systems and for modified systems.

Although **UNIX United** uses the same mechanism for communications (sockets), it employs the datagram service which does not guarantee reliable delivery of messages. In benchmark testing our group found, and Mr. Donnelly's paper mentions the fact, that datagram service is slower than stream service. In **UNIX United** a separate layer of software is provided to perform ordering and eliminate duplicate messages. The **DTroll** communications programs take advantage of the speed of stream service by allowing a user to specify such service. However, the user may elect datagram service instead, but there are no provisions at present to handle the problems associated with this service, such as lost messages, duplicate messages, or messages out of order.

Another disadvantage of the **UNIX United** system was reliability. On each host supported there exists a process called a **UNIX server activation manager** or **USAM**.

It is responsible for intercepting remote requests and spawning a service routine. If this process dies or hangs, which during our testing was very frequent, the system does not work. There are no watch dogs in the system to alert the system to restart the process. Thus it can be a long period of time before the outage is noted by personnel authorized to restart it. In the **DTroll** system the various **Status Monitors** and **Mother** processes act as watch dogs. In the event of a site outage, the **Status Monitor**, after a timeout waiting for an answer from another **Status Monitor**, can check with the system through the **ruptime** facility to double check whether the site is down. It can then take action to get the appropriate routines running on the correct host, if site outage is not the problem.

With the proper linking and other actions, **UNIX United** could support the communications needs of the **DTroll** system. However, the strong lack of confidence by members of the **DTroll** group in **UNIX United**, the improved performance and control realized by **DTroll** oriented software, and a desire to avoid tying the **DTroll** system to only systems supporting and operating with **UNIX United** prompted the development of the communications facilities **INITSERVSOCKET** and **RQSTCONNECT** within the **DTroll** system.

These communications facilities are the topic of the remainder of this paper.

CHAPTER 2

INITSERVSOCKET

2.1. Purpose and Parameters

The subroutine **INITSERVSOCKET** is used for server processes to establish their communications connectors. If successful, the routine will have established and conditioned a communications channel and be awaiting a connection by a client process. The format of the call is:

INITSERVSOCKET(**sname, domain, type, proto, blog);**

where

Sname is the formal name used to identify the particular server requesting the channel. The format of this parameter changes based on the configuration of the system as governed by the value of **MODE** or the **domain** parameter. **MODE**, **AF_UNIX**, and **AF_INET** are constants explained later in this chapter.

The formats for **sname** are:

(1). **<server name><backslash 0>**

(2). **<port number>:<host name>:<server name><backslash 0>**

(3). **<host name>:<server name><backslash 0>**

If the **domain** parameter is equal to **AF_UNIX**, format (1) is assumed. In this case, communications will be local to the host machine and the necessity of providing the

additional information necessary to establish a network addressing relationship is eliminated.

If the **domain** parameter is equal to **AF_INET** and **MODE** is equal to "1" format (2) is assumed. In this case, the identifiers for the **DTroll** server names are not present in the host/network tables. Therefore,, the logical **port number** which is to be associated with the channel is required along with other network addressing information. The **host name** can be either the primary identifier or any of the valid alias names by which the host is known within the system. **Server name** is the formal identifier of the server requesting the channel. Due to the application of string manipulation routines on the **sname** parameter, the string must be terminated with a backslash 0. In keeping with the convention of system modules such as **rsh**, ":" is used to delimit fields and is mandatory. The parameter **sname** should contain no embedded blanks.

If the **domain** parameter is equal to **AF_INET** and **MODE** is equal to "0", format (3) is assumed. In this case, information regarding the **DTroll** servers is contained in the host/network tables. The **port number** is not required but all other information, as described for format (2), is the same.

The **domain** parameter identifies the arena in which communications will take place. Currently, the system will support **AF_UNIX** for intra-machine linkages and **AF_INET** for inter-machine linkages. Note: The use of **AF_INET**, although possibly less efficient, can be used for intra-machine linkage as well.

The **type** parameter describes the type of service offered. Currently supported is **SOCK_STREAM**, which guarantees bidirectional, reliable, sequenced, and unduplicated flow of data. Also supported is **SOCK_DGRAM**, which supports bidirectional flow of

data but does not promise sequenced, reliable, nor unduplicated data. For the **DTroll** application, it is suggested that only the **SOCK_STREAM** option be used at present.

The parameter **proto** is the protocol used during communications. For the present, **TCP** should be used.

The parameter **blog** is the number of requests for connection we wish the system to queue and manage for the channel. At this time, this is limited to a maximum of five.

In the original version of **INITSERVSOCKET**, **sname** was the only parameter required. However, to allow for future expansion of the interprocess communications facilities [LFJo,83], the additional parameters were included to provide the user greater flexibility.

If a call to **INITSERVSOCKET** is successful, a non–negative integer, analogous to a file descriptor, is returned to the caller. This descriptor can then be used with either an **accept** or **select** system call to effect the communications process. Data is passed on an established socket primarily by the **read** and **write** system calls. If a call is unsuccessful, a negative value will be returned to indicate an error.

Once a socket has been successfully established and used, the descriptor, and thus the socket, can be eliminated using the **close** system call. This code, although designed for use within the **DTroll** system, can be used in other applications with slight modification. The system files describing constants such as **AF_UNIX**, **TCP**, etc.; data structures such as **hostent**, **sockaddr**, etc; and system call primitives, are contained within a **DTroll** header file. To incorporate **INITSERVSOCKET** or **RQSTCONNECT** (**Chapter 3**) modules, one must include the system files:

sys/types.h, **sys/socket.h**, **netdb.h**, **netinet/in.h**, and **sys/un.h**.

2.2. Predefined System Entities

There are several predefined structures used in this module. The structures are accessed and manipulated by the various system calls such as **socket**, **bind**, etc. To provide the user with a greater understanding, a brief description of each is provided below. Also included are descriptions of some of the predefined constants used within this paper.

AF_UNIX is a predefined constant which is set to "1". It defines the arena of communications operations to be local. When the **domain** parameter is assigned the value **AF_UNIX**, all communications take place between processes on the same host.

AF_INET is a predefined constant which is set to "2". It defines the arena of communications operations as an internetwork environment. When the **domain** parameter is assigned the value **AF_INET**, for the **DTroll** system, transactions may occur between VAX host machines on the department network.

SOCK_STREAM is a predefined constant which is set to "1". It defines the actual type of service to be "stream". This establishes a bidirectional, reliable, sequenced and unduplicated flow of data across the communications channel. The **type** parameter is assigned the value **SOCK_STREAM** to select "stream" type service.

TCP is a predefined constant which is set to "1". It defines the protocol to be used during communications operations to be **TCP**. This acronym stands for **Transmission Control Protocol**. It is a protocol standard adopted by the Department of Defense. The **proto** parameter is assigned the value **TCP** to indicate that **Transmission**

Control Protocol be used.

The structure type, **sockaddr**, contains information regarding the socket. It consists of two fields. The **sa_family** field contains an integer used to identify the arena in which communications will take place (**AF_UNIX** or **AF_INET**). The **sa_data** field is an array of fourteen bytes used to hold the direct address of the socket.

The structure type, **hostent**, contains information on the host. It consists of five fields. The **h_name** field contains the official name of the host. The **h_aliases** field is a list of alias names. The **h_addrtype** field is the host address type of either a local or specific network. The **h_length** field contains the length (in bytes) of the address in the **h_addr** field. The **h_addr** field is the actual address of the host.

The structure type, **servent**, contains information regarding a server process. It contains four fields. The **s_name** field is the official name of the server as listed in the host/network system tables. The **s_aliases** field is a list of alias names by which the service can be addressed. The **s_port** field holds the port number to which the service is tied or associated. The **s_proto** field identifies the protocol to be used. This structure is stocked with information through the **getservbyname** system call within the program.

2.3. Constants

INITSERVSOCKET uses global constants defined outside the module code. These are:

```
#define MODE 1
#define HOST_ERR -1
#define SOCK_ERR -2
```

#define SERV_ERR -3

#define BIND_ERR -4

#define LSTN_ERR -5

#define DOMN_ERR -6

MODE is a configuration flag. The value "1" represents the current configuration whereby the names of the **DTroll** server processes are not contained within the host/network tables. Therefore, the system routine **getservbyname** can not be used to gain information about servers. Instead this information is to be supplied by the user through the **sname** parameter.

Upon demonstration that the system is performing reliably and is safe from a security standpoint, the administrator of the network/host systems will enter the **DTroll** server names in the host/network system tables and associate with each a privileged port number. A server will always listen at this well-known port number for client connections. Once this action is accomplished, the **MODE** should be set to "0". This will relieve the user of the requirement to include a port number in the **sname** parameter. The constants ending in **_ERR** are the error codes returned to the calling program in the event of a failure during the process of establishing a communications connector.

HOST_ERR indicates a problem in finding or constructing the host address. The host name must be either the primary identifier or a valid alias.

SOCK_ERR indicates a problem in obtaining a socket from the system.

SERV_ERR indicates a problem in finding the service name in the system tables. When **MODE** is set to "1", this should never be a valid error as the code does not call

getservbyname to obtain information contained in the system tables. **MODE** "1" indicates (as is the current situation) that there are no entries for the **DTroll** service name in the host/network system tables.

BIND_ERR indicates the binding of the socket to either a **UNIX** type path name (**AF_UNIX**) or an address space (**AF_INET**) has failed.

LSTN_ERR indicates a problem in configuring the port with the **listen** command.

DOMN_ERR indicates the domain supplied is not supported. At the present only the predefined types **AF_UNIX** or **AF_INET** are supported by the **DTroll** system.

2.4. Error Codes

If a call to **INITSERVSOCKET** is unsuccessful, a negative number is returned. The following list of values will help the user to localize the type of error encountered. For more specific information on the error which has occurred, one should use the system global variable **errno** and/or the system call **perror**. The codes applicable to **INITSERVSOCKET** are:

–1 Indicates a problem finding or constructing the host address. The host name must be either the primary identifier or a valid alias.

–2 Indicates a problem in obtaining a socket from the system.

–3 Indicates a problem in finding the service name in the system tables. When **MODE** is set to "1", this should never be a valid error. **MODE** "1" indicates (as is the current situation) that there are no entries for the **DTroll** service name in the host/network system tables.

−4 Indicates a problem in binding the socket to either a path name entity (**AF_UNIX**) or an address space (**AF_INET**). The socket, once established, must be bound by the server process to the entity corresponding to the environment in which it was created.

−5 Indicates a problem in opening the socket for the server to receive a request for communications from a client process. This is referred to as "listening".

−6 Indicates the domain supplied is not supported. At present the predefined types **AF_UNIX** or **AF_INET** are supported.

2.5. Detailed Code Description

The following is a detailed description of the source code for **INITSERVSOCKET** which can be found in **Appendix A**. The constants for this module are described in section 2.3.

Lines 1–5: Subroutine and parameter definition. Parameters are described in detail above.

Line 7: The variable **s** is used to contain and return the socket descriptor number.

Line 8: The variables **i** and **j** are used as indices during the disassembly of the sname parameter into the various informational fields.

Line 9: The array **host** is used to hold the name of the host machine encoded in the **sname** parameter.

Line 10: The array **server** is used to hold the name of the server encoded in the **sname** parameter.

Line 11: The variable **portal** is used to hold the port number encoded in **sname**. This is only used if the **MODE** is "1".

Line 12: The pointer **hp** points to a predefined structure (**hostent type**) which will hold the pertinent information about the host passed in **sname**.

Lines 13–14: The structures **sin** (**sockaddr_in type** for an internetwork environment) and **sun** (**sockaddr_un type** for a local environment) are used to hold all applicable information about the communications connector including addressing information. Like the structure pointed to by **hp**, these structures are predefined, meaning their definition is contained in the files linked at compile time.

Line 15: The pointer **sp** (**servent type**) points to another predefined structure which holds the information found in the system tables concerning a server. When **MODE** is "1" this information is not gathered, but is assembled from data provided explicitly by the calling program in the **sname** parameter.

Line 18: Since the operations performed to establish a communications connector are primarily dependent on the operating environment, a switch is made to perform the connector setup based on the **domain** parameter.

Line 19: The first case handled is that of inter-machine linkage (**AF_INET**).

Line 21: **I** is initialized to the first position of **sname**. **I** is maintained as a pointer into the parameter **sname** to define the current position. Therefore, it is cumulative and is not reinitialized.

Line 22: **Portal** is initialized (in the event it is necessary) to "0".

Line 23: **MODE** is tested to decide if the port number is included in the **sname** parameter.

Line 25: If **MODE** was "1", the port number is removed from **sname**, digit by digit, until the delimiter is encountered.

Line 27: As each digit is stripped from **sname**, the actual value obtained is the ASCII character representation of the digit. Therefore, by multiplying **portal** by "10" and adding the character representation minus "48" (the actual offset between the ASCII character representation and the numeric value of the digit) the port number in numeric form is built.

Line 28: **I** is incremented to the next position in **sname**.

Line 30: When the port number has been extracted, **i** is incremented to skip over the delimiter.

Line 32: **J** is initialized to the first element of the **host** array.

Line 33: The informational field is stripped out until a delimiter is encountered.

Lines 35–37: A direct transfer of information is effected by stepping **i** and **j** up on each iteration.

Lines 39–40: When the host name has been extracted, **i** is incremented to step over the delimiter and **j** is initialized to point to the first element of the **server** array.

Line 41: The server name is the last informational field contained in **sname**. It is transferred to the **server** array until the backslash 0 is encountered.

Lines 43–45: A direct transfer of information is effected by stepping **i** and **j** up on each iteration.

Lines 47–48: A call to **gethostbyname** is invoked. If successful, the pointer **hp** will point to a structure containing the information obtained from system tables. Contained in this information is the formal name by which the host is known to the network and other hosts. If an error is encountered, the routine is terminated and the appropriate error code returned.

Line 49: The call to **bzero** is used to initialize the structure to hold the socket information. **Bzero** places length "0" bytes (given by its second parameter) into the string represented by its first argument.

Line 50: The address of the host (given in the first argument) is transferred to the socket structure field (given in the second argument) by a call to **bcopy**. The number of bytes transferred is given in the third argument.

Line 51: The socket structure **family** field is set to **AF_INET**.

Lines 52–53: If **MODE** is "1", the port number was obtained from the parameter **sname** and only requires conversion to the network compatible form. This conversion is performed by a call to **ntons**.

Lines 54–58: If **MODE** is not equal to "1", the port number will be obtained from the system tables. The call to **getservbyname** searches the table for a match and fills the server structure. If the call is unsuccessful, the routine is terminated and the proper error code returned. If the call is successful, line 58 transfers the information from the server structure **port** field to the socket **port** field.

Lines 59–60: If a successful call to **socket** is made, a socket identifier, configured for the **domain**, **type**, and **protocol** passed to the routine, is returned. If unsuccessful, the routine is terminated and the error code for a socket error is returned.

Lines 61–62: Once the socket is obtained, it must be bound to the address space for the server. If the call to **bind** is successful, the routine continues. If it is unsuccessful, the error code for a binding error is returned.

Lines 63–66: The final preparation of the socket is to open it for acceptance of requests. This is accomplished with a call to **listen**. If successful, the process is complete and the descriptor for the valid socket is returned. If the call is unsuccessful, the error code for a listening error is returned.

Line 68: This marks the end of the **AF_INET** case statements.

Line 69: If the domain is **AF_UNIX**, a much simpler linking process is followed.

Line 72: Since the **AF_UNIX** domain associates the socket address relation with the **UNIX** style path name, and this descriptor is placed in the user space, it is necessary to release any file or descriptor by this name prior to the creation of a new entity. This housekeeping chore is performed by a call to **unlink**.

Line 73: The socket structure **family** field is set to **AF_UNIX**.

Line 74: The server name, in **sname**, is copied to the socket structure **path** field by a call to **strcpy**.

Lines 75–76: If the call to **socket** is successful, a communications connector descriptor is returned. If it is unsuccessful, the socket error code is returned.

Lines 77–78: Once a valid socket is obtained, it must be bound to a **UNIX** style path name. A successful call to **bind** will accomplish this task. The convention chosen is to bind to the server name provided through the **sname** parameter. If the call is unsuccessful, the error code for the bind is returned.

Lines 79–82: The final preparation of the socket is to open it to accept requests. This is accomplished through a call to **listen**. If successful, the descriptor number is returned to the caller. If unsuccessful, the error code for the listen is returned.

Line 84: This marks the end of the **AF_UNIX** case statements.

Lines 85–87: If the **domain** is incorrect, the routine is terminated and the proper error code returned.

CHAPTER 3

RQSTCONNECT

3.1. Purpose and Parameters

The subroutine **RQSTCONNECT** is used to establish a client's communications connection with a designated server process. If successful, the routine will have established the connection to the server and the using program may then commence to transfer or receive information across this communications channel. Much of the procedure is very similar to **INITSERVSOCKET**, although in general this is a simpler set of events. The format of the call is:

RQSTCONNECT(sname, domain, type, proto)

where

Sname is the formal name used to identify the particular server to which we wish to connect. The format of this parameter changes based on the configuration of the system as governed by the value of **MODE** or the **domain** parameter. **MODE**, **AF_UNIX**, and **AF_INET** were explained in **Chapter 2**. Explanations of these and other predefined entities are duplicated in this chapter (Sections 3.2 & 3.3) to make it self contained.

The formats for **sname** are:

(1). **<server name><backslash 0>**

(2). **<port number>:<host name>:<server name><backslash 0>**

(3). **<host name>:<server name><backslash 0>**

If the **domain** parameter is equal to **AF_UNIX**, format (1) is assumed. In this case, communications will be local to the host machine and the necessity of providing the additional information necessary to establish a network addressing relationship is eliminated.

If the **domain** parameter is equal to **AF_INET** and **MODE** is equal to "1", format (2) is assumed. In this case, the identifiers for the **DTroll** server names are not present in the host/network system tables. Therefore, the logical **port number** to which connection is associated is required along with other network addressing information. The **host name** can be either the primary identifier or any of the valid alias names by which the host is known within the system. **Server name** is the formal identifier of the server. Due to the application of string manipulation routines on the **sname** parameter, the string must be terminated with a backslash 0. In keeping with the convention of system modules such as **rsh**, ":" is used to delimit fields and is mandatory. The parameter **sname** should contain no embedded blanks.

If the **domain** parameter is equal to **AF_INET** and **MODE** is equal to "0", format (3) is assumed. In this case, information regarding the **DTroll** servers is contained in the host/network tables. The **port number** is not required and all other information, as described for format (2), is the same.

The **domain** parameter identifies the arena in which the communications will take place. Currently, the system will support **AF_UNIX** for intra-machine linkages and

AF_INET for inter-machine linkages. Note: The use of **AF_INET**, although possibly less efficient, can be used for intra-machine linkage as well.

The **type** parameter describes the type of service offered. Currently supported is **SOCK_STREAM** which guarantees bidirectional, reliable, sequenced, and unduplicated flow of data. Also supported is **SOCK_DGRAM** which supports bidirectional flow of data but does not promise sequenced, reliable, nor unduplicated data. For the **DTroll** application, it is suggested that only the **SOCK_STREAM** option be used at present.

The parameter **proto** is the protocol used during communications. For the present, **TCP** should be used.

When the socket becomes obsolete, it can be eliminated with a call to **close**.

As with **INITSERVSOCKET**, if the call is a success, a non-negative integer, analogous to a file descriptor, is returned to the caller. Unlike **INITSERVSOCKET**, no further manipulation of this socket is necessary to effect communications. The caller may send or receive data on the established socket through the use of the **read** and **write** system calls. If a call is unsuccessful, a negative value will be returned to indicate an error.

As with **INITSERVSOCKET**, this procedure may be used in other applications provided **sys/types.h**, **sys/socket.h**, **netdb.h**, **netinet/in.h**, and **sys/un.h** system files are linked in at compilation time.

3.2. Predefined System Entities

There are several predefined structures used in this module. The structures are accessed and manipulated by the various system calls such as **socket**, **bind**, etc. To provide the user with a greater understanding a brief description of each is provided below. Also included are descriptions of some of the predefined constants used within this paper.

AF_UNIX is a predefined constant which is set to "1". It defines the arena of communications operations to be local. When the **domain** parameter is assigned the value **AF_UNIX**, all communications take place between processes on the same host.

AF_INET is a predefined constant which is set to "2". It defines the arena of communications operations as an internetwork environment. When the **domain** parameter is assigned the value **AF_INET**, for the **DTroll** system, transactions may occur between VAX host machines on the department network.

SOCK_STREAM is a predefined constant which is set to "1". It defines the actual type of service to be "stream". This establishes a bidirectional, reliable, sequenced and unduplicated flow of data across the communications channel. The **type** parameter is assigned the value **SOCK_STREAM** to select "stream" type service.

TCP is a predefined constant which is set to "1". It defines the protocol to be used during communication operations to be **TCP**. This acronym stands for **Transmission Control Protocol**. It is a protocol standard adopted by the Department of Defense. The **proto** parameter is assigned the value **TCP** to indicate that **Transmission Control Protocol** be used.

The structure type, **sockaddr**, contains information regarding the socket. It consists of two fields. The **sa_family** field contains an integer used to identify the arena in which communications will take place (**AF_UNIX** or **AF_INET**). The **sa_data** field is an array of fourteen bytes used to hold the direct address of the socket.

The structure type, **hostent**, contains information on the host. It consists of five fields. The **h_name** field contains the official name of the host. The **h_aliases** field is a list of alias names. The **h_addrtype** field is the host address type of either a local or specific network. The **h_length** field contains the length (in bytes) of the address in the **h_addr** field. The **h_addr** field is the actual address of the host.

The structure type, **servent**, contains information regarding a server process. It contains four fields. The **s_name** field is the official name of the server as listed in the host/network system tables. The **s_aliases** field is a list of alias names by which the service can be addressed. The **s_port** field holds the port number to which the service is tied or associated. The **s_proto** field identifies the protocol to be used. This structure is stocked with information through the **getservbyname** system call within the program.

3.3. Constants

RQSTCONNECT uses global constants defined outside the module code. These are:

```
#define MODE 1
#define HOST_ERR -1
#define SOCK_ERR -2
#define SERV_ERR -3
#define DOMN_ERR -6
```

#define CNCT_ERR -7

MODE is a configuration flag. The value "1" represents the current configuration whereby the names of the **DTroll** server processes are not contained within the host/network tables. Therefore, the system routine **getservbyname** can not be used to gain information about servers. Instead this information (explained later) needs to be supplied by the user.

Upon demonstration that the system is performing reliably and is safe from a security standpoint, the administrator of the network/host systems will enter the **DTroll** server names in the host/network system tables and associate with each a privileged port number. A server will always listen at this well-known port number for client connections. Once this action is accomplished, the **MODE** should be set to "0". This will relieve the user of the requirement to include a port number in the **sname** parameter. The constants ending in **_ERR** are the error codes returned to the calling program in the event of a failure during the process of establishing a communications connector.

HOST_ERR indicates a problem finding or constructing the host address. The host name must be either the primary identifier or a valid alias.

SOCK_ERR indicates a problem in obtaining a socket from the system.

SERV_ERR indicates a problem in finding the service name in the system tables. When **MODE** is set to "1", this should never be a valid error as the code does not call **getservbyname** to obtain information contained in the system tables. **MODE** "1" indicates (as is the current situation) that there are no entries for the **DTroll** service name in the host/network system tables.

DOMN_ERR indicates the domain supplied is not supported. At the present only the predefined types **AF_UNIX** or **AF_INET** are supported by the **DTroll** system.

CONT_ERR indicates the linkage to the server through the **connect** call has failed. One of the primary reasons for such a failure is the server not having a connector open and configured through the **accept** system call. Another reason for failure is the server being down.

3.4. Error Codes

If a call to **RQSTCONNECT** is unsuccessful, a negative number is returned. The following returned values will help to localize the type of error encountered. For more specific information on the error which has occurred, one should use the system global variable **errno** and/or the system call **perror**. The codes applicable to **RQSTCONNECT** are:

−1 Indicates a problem finding or constructing the host address. The host name must be either the primary identifier or a valid alias.

−2 Indicates a problem in obtaining a socket from the system.

−3 Indicates a problem in finding the service name in the system tables. When **MODE** is set to "1", this should never be a valid error. **MODE** "1" indicates (as is the current situation) that there are no entries for the **DTroll** service name in the host/network system tables.

−6 Indicates the domain supplied is not supported. The common error is misspelling of the predefined types **AF_UNIX** or **AF_INET**.

−7 Indicates the linkage to the server through the **connect** call has failed. One of the primary reasons for such a failure is the server not having a connector open and configured through the **accept** system call. Another reason for failure is the server being down.

3.5. Detailed Code Description

The following is a detailed description of the source code for **RQSTCONNECT** which can be found in Appendix B. The constants for this module are described in **Chapter 1**.

Lines 1–5: Subroutine and parameter definition. Parameters are described in detail above.

Line 7: The variable **s** is used to contain and return the socket descriptor number.

Line 8: The variables **i** and **j** are used as indices during the disassembly of the **sname** parameter into the various informational fields.

Line 9: The array **host** is used to hold the name of the host machine encoded in the **sname** parameter.

Line 10: The array **server** is used to hold the name of the server encoded in the **sname** parameter.

Line 11: The variable **portal** is used to hold the port number encoded in **sname**. This is only used if the **MODE** is "1".

Line 12: The pointer **hp** points to a predefined structure (**hostent type**) which will hold the pertinent information about the host passed in **sname**.

Lines 13-14: The structures **sin** (**sockaddr_in type** for an internetwork environment) and **sun** (**sockaddr_un type** for a local environment) are used to hold all applicable information about the communications connector including addressing information. *Like the structure pointed to by* **hp**, *these structures are* predefined, meaning their definition is contained in the files linked in at compile time.

Line 15: The pointer **sp** (**servent type**) points to another predefined structure which holds the information found in the system tables concerning a server. When **MODE** is "1" this information is not gathered, but is assembled from data provided explicitly by the calling program in the **sname** parameter.

Lines 17-18: If a successful call to **socket** is made, a socket identifier configured for the **domain**, **type**, and **protocol** passed to the routine is returned. If unsuccessful, the routine is terminated and the error code for a socket error is returned.

Line 20: Since the operations performed to establish a communications connector are primarily dependent on the operating environment, a switch is made to perform the connector setup based on the **domain** parameter.

Line 21: The first case handled is that of inter-machine linkage (**AF_INET**).

Line 23: I is initialized to the first position of **sname**. I is maintained as a pointer into the parameter **sname** to define the current position. Therefore, it is cumulative and is not reinitialized.

Line 24: **Portal** is initialized (in the event it is necessary) to "0".

Line 25: **MODE** is tested to decide if the port number is included in the **sname** parameter.

Line 27: If **MODE** was "1", the port number is removed from **sname**, digit by digit, until the delimiter is encountered.

Line 29: As each digit is stripped from **sname**, the actual value obtained is the ASCII character representation of the digit. Therefore, by multiplying **portal** by "10" and adding the character representation minus "48" (the actual offset between the ASCII character representation and the numeric value of the digit) the port number in numeric form is built.

Line 30: **I** is incremented to the next position in **sname**.

Line 32: When the port number has been extracted, **i** is incremented to skip over the delimiter.

Line 34: **J** is initialized to the first element of the **host** array.

Line 35: The informational field is stripped out until a delimiter is encountered.

Lines 37–39: A direct transfer of information is effected by stepping **i** and **j** up on each iteration.

Lines 41–42: When the host name has been extracted, **i** is incremented to step over the delimiter and **j** is initialized to point to the first element of the **server** array.

Line 43: The server name is the last informational field contained in **sname**. It is transferred to the **server** array until the backslash 0 is encountered.

Lines 45–47: A direct transfer of information is effected by stepping **i** and **j** up on each iteration.

Lines 49–50: A call to **gethostbyname** is invoked. If successful, the pointer **hp** will point to a structure containing the information obtained from system tables. Contained in this information is the formal name by which the host is known to the network and other hosts. If an error is encountered, the routine is terminated and the appropriate error code returned.

Line 51: The call to **bzero** is used to initialize the structure to hold the socket information. **Bzero** places length "0" bytes (given by its second parameter) into the string represented by its first argument.

Line 52: The address of the host (given in the first argument) is transferred to the socket structure field (given in the second argument) by a call to **bcopy**. The number of bytes transferred is given in the third argument.

Line 53: The socket structure **family** field is set to **AF_INET**.

Lines 54–55: If **MODE** is "1", the port number was obtained from the parameter **sname** and only requires conversion to the network compatible form. This conversion is performed by a call to **ntons**.

Lines 56–60: If **MODE** is not equal to "1", the port number will be obtained from the system tables. The call to **getservbyname** searches the table for a match and fills the server structure. If the call is unsuccessful, the routine is terminated and the proper error code returned. If the call is successful, line 60 transfers the information from the server structure **port** field to the socket **port** field.

Lines 61–64: If the call to **connect** is unsuccessful, the routine is terminated and the proper error code is returned. However, if it is successful, the descriptor number for

the communications connector is returned.

Line 66: This marks the end of the **AF_INET** case statements.

Line 67: If the domain is **AF_UNIX**, a much simpler linking process is followed.

Line 70: Since the **AF_UNIX** domain associates the socket address relation with the**UNIX** style path name and this descriptor is placed in the user space, it is necessary to release any file or descriptor by this name prior to the creation of a new entity This housekeeping chore is performed by a call to **unlink**.

Line 71: The socket structure **family** field is set to **AF_UNIX**.

Line 72: The server name, in **sname**, is copied to the socket structure **path** field by a call to **strcpy**.

Lines 73–76: If the call to **connect** is unsuccessful, the routine is terminated and the proper error code is returned. If successful, the descriptor number for the communications connector is returned.

Line 78: This marks the end of the **AF_UNIX** case statements.

Lines 79-81: If the domain is incorrect, the routine is terminated and the proper error returned.

CONCLUSION

The communications software described in this paper is currently being used by many of the **DTroll** system modules to establish their communications connections.

It will be necessary, if datagram service is desired in the future, to develop the software to check for lost messages, duplicate messages, and messages delivered out of order. Datagram service may be desirable for updates to multiple sites containing replicated data. As long as the system remains relatively small, this can be accomplished by addressing individual sites in separate messages.

Another aspect of communications, the interface between user and system (query language) is under development. It is taking advantage of the system tools **YACC** (**Yet Another Compiler Compiler**) and **Lex** (a lexical analyzer builder).

Finally, individual protocols between processes within the **DTroll** system are being established by the person or persons developing the routines. Details and explanations of these protocols will be contained in their published works.

APPENDIX A

INITSERVSOCKET

```
INITSERVSOCKET(sname,domain,type,proto,blog)

char sname[];
int domain,type,proto,blog;

{
        register int s;
        int i,j;
        char host[30];
        char server[256];
        int portal;
        struct hostent *hp;
        struct sockaddr_in sin;
        struct sockaddr_un sun;
        struct servent *sp;

```

```
switch(domain){
case AF_INET:
        {
                i = 0;
                portal = 0;
                if (MODE == 1)
                {
                        while(sname[i] != ':')
                        {
                                portal = (portal * 10) + (((int)sname[i])-48);
                                i++;
                        }
                        i++;
                }
                j = 0;
                while(sname[i] != ':')
                {
                        host[j] = sname[i];
                        i++;
```

```
        j++;
    }
    i++;
    j = 0;
    while(sname[i] != '\0')
    {
        server[j] = sname[i];
        i++;
        j++;
    }
    if ((hp = gethostbyname(host)) == NULL)
        return(HOST_ERR);
    bzero((char *)&sin,sizeof(sin));
    bcopy(hp->h_addr,(char *)&sin.sin_addr,hp->h_length);
    sin.sin_family = AF_INET;
    if(MODE == 1)
        sin.sin_port = htons(portal);
    else
        if ((sp = getservbyname(server,proto)) ==
           (struct servent *) NULL)
```

```
                                return(SERV_ERR);
                        else
                                sin.sin_port = sp ->s_port;
                if ((s= socket(domain,type,0))<= -1)
                        return(SOCK_ERR);
                if (bind(s, (caddr_t)&sin, sizeof(sin)) < 0)
                        return(BIND_ERR);
                if (listen(s,blog) < 0)
                        return(LSTN_ERR);
                else
                   return(s);
        }
        break;
case AF_UNIX:
        {

                unlink(sname);
                sun.sun_family = AF_UNIX;
                strcpy(sun.sun_path, sname);
                if ((s=socket(domain,type,0))<= -1)
```

```
                return(SOCK_ERR);
            if (bind(s, (struct sockaddr *)&sun,
              strlen(sun.sun_path) + 2) < 0)
                return(BIND_ERR);
            if (listen(s,blog) < 0)
                return(LSTN_ERR);
            else
              return(s);
        }
        break;
    default:
        return(DOMN_ERR);
        break;
    }
}
```

APPENDIX B

RQSTCONNECT CODE

```
RQSTCONNECT(sname,domain,type,proto)

char sname[];
int domain,type,proto;

{
        register int s;
        int i,j;
        char host[30];
        char server[256];
        int portal;
        struct hostent *hp;
        struct sockaddr_in sin;
        struct sockaddr_un sun;
        struct servent *sp;

```

```
if((s = socket(domain,type,proto)) < 0)
        return(SOCK_ERR);

switch(domain){
case AF_INET:
        {
                i = 0;
                portal = 0;
                if (MODE == 1)
                {
                        while(sname[i] != ':')
                        {
                                portal = (portal * 10) + (((int)sname[i])-48);
                                i++;
                        }
                        i++;
                }
                j = 0;
                while(sname[i] != ':')
                {
```

```
        host[j] = sname[i];
        i++;
        j++;
    }
    i++;
    j = 0;
    while(sname[i] != '\0')
    {
        server[j] = sname[i];
        i++;
        j++;
    }
    if ((hp = gethostbyname(host)) == NULL)
        return(HOST_ERR);
    bzero((char *)&sin,sizeof(sin));
    bcopy(hp->h_addr,(char *)&sin.sin_addr,hp->h_length);
    sin.sin_family = AF_INET;
    if(MODE == 1)
        sin.sin_port = htons(portal);
    else
```

```
                        if ((sp = getservbyname(server,proto))
                         == (struct servent *) NULL)
                                return(SERV_ERR);
                        else
                                sin.sin_port = sp->s_port;
                if(connect(s,(char *)&sin,sizeof(sin)) < 0)
                        return(CNCT_ERR);
                else
                        return(s);
        }
        break;
case AF_UNIX:
        {

                unlink(server);
                sun.sun_family = AF_UNIX;
                strcpy(sun.sun_path, server);
                if(connect(s,(char *)&sin,sizeof(sin)) < 0)
                        return(CNCT_ERR);
                else
```

```
                        return(s);
                }
                break;
        default:
                return(DOMN_ERR);
                break;
        }
}
```

REFERENCES

[Mage,86] P. J. Magelli Jr., DTROLL: The Design of a Research Distributed Database System, M.S. University of Illinois, (1986).

[Donn,85] J. M. Donnelly, Porting The Newcastle Connection to 4.2 Berkeley UNIX, M.S. University of Illinois, (1985).

[Goth,85] R. E. Gothelf. UNIX United in Development of Dtroll, M.C.S. University of Illinois. (1985).

[JKim,85] J. K. Kim. The Dtroll System. University of Illinois, Urbana, Illinois, (August 1985).

[UNIX,84] The UNIX Programmer's Manual, 4.2 Berkeley Software Distribution. Virtual VAX-11 Version. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Ca (March 1984).

[LFJo,83] S. J. Leffler, R. S. Fabry and W. N. Joy. A 4.2bsd Interprocess Communication Primer. Computer Systems Research Group, Department of Electical Engineering and Computer Science, University of California, Berkely, Ca, (March 1983).

ATE

LMED

8